Towards modelling WS-BPEL using WS-CALCULUS

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica,
Università degli Studi di Firenze,
Viale Morgagni, 65 - I-50134 Firenze - Italy

Abstract. We tackle the problem of providing rigorous formal foundations to current software engineering technologies for web services, and especially to WS-BPEL, one of the most used XML-based standard languages for web services. We focus on a subset of WS-BPEL sufficiently expressive to model the interactions among web service instances in a network context. We present this language as a process calculus-like formalism, that we call WS-CALCULUS, for which we define an operational semantics. A pair of illustrative examples illustrates WS-CALCULUS peculiarities and show its expressiveness. Finally, we extend the WS-CALCULUS to define an operational semantics for a more complete set of WS-BPEL functionalities.

Keywords:
Web services; orchestration languages; WS-BPEL; formal methods;

Contents

1. Introduction

Service-Oriented Computing (SOC) has recently put forward as a promising computing paradigm for developing massively distributed, interoperable, evolvable systems and applications that exploit the pervasiveness of the Internet and its related technologies. The SOC paradigm advocates the use of 'services', to be understood as autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled, as the basic blocks for building applications. Web services (WSs), along with grid computing, are the present most successful instantiation of the SOC paradigm, as it is demonstrated by the fact that companies like IBM, Microsoft and Sun invested a lot of efforts and resources to promote their deployment.

A *web service* is basically a set of operations that can be invoked through the Web via XML messages complying with given standard formats. To support the WS approach, many new languages, most of which based on XML, have been designed, like e.g. business coordination languages (such as WS-BPEL, WSFL, WSCI, and XLANG), contract languages (such as WSDL and SWS), and query languages (such as XPath and XQuery). However, current software engineering technologies for WS still lack rigorous formal foundations. The challenges come from the necessity of dealing at once with issues like asynchronous interactions, concurrency, workflow coordination, business transactions, resource usage, failures, security, etc. in a setting where demands and guarantees can be very different for the many components.

We focus on one of the most used XML-based languages for WSs: *Web Services Business Process Execution Language* (WS-BPEL [OAS07]), an OASIS standard that permits to describe the business logic and the interactions to be executed for completing the service as a reaction to a service invocation. The service often results from the *orchestration* of other available services, i.e. from their aggregation and invocation according to a given set of rules to meet a business requirement. In conclusion, WS-BPEL descriptions can be used to define new services by appropriately orchestrating other existing ones.

In this paper, we first define a semantic model for WS-BPEL because the semantics of the language, as presented in [OAS07], is informal and, sometimes, ambiguous. Hence, as a first contribution of this paper, we introduce a process language, that we call WS-CALCULUS (*web services calculus*), that formalizes the semantics of an expressive subset of WS-BPEL, with special concern for modelling the interactions among web services, be them WS-BPEL processes or not, in a network context. This allows us to tackle those problems arising when executing WS-BPEL processes, such as multiple start activities, receive conflicts, routing of messages, while avoiding the intricacies of dealing with any, possibly redundant, WS-BPEL construct.

Then, we extend the WS-CALCULUS to incorporate such other important constructs of WS-BPEL as flow graphs, timed activities, scopes and compensation handling, that should be considered for modelling the semantics of full-blown orchestration languages. We will show that flow graphs can be easily encoded in WS-CALCULUS, while the semantics of the remaining constructs is given by extending the operational semantics of WS-CALCULUS. This way, as a second contribution, we define an operational semantics for WS-BPEL *executable* processes.

Overview of the rest of the paper. Syntax and operational semantics of WS-CALCULUS are defined in Section 2. Section 3, by means of a shipping service scenario borrowed from the official WS-BPEL specification, shows a WS-BPEL program and the corresponding WS-CALCULUS specification. Section 5 extends WS-CALCULUS and defines an operational semantics for the whole WS-BPEL. In Section 6 we conclude by touching upon directions for future work and comparisons with related work.

2. WS-CALCULUS

This section introduces syntax and semantics of WS-CALCULUS (*web services calculus*). The calculus can directly model the semantics of an expressive subset of WS-BPEL. Indeed, we will see that it permits to express web services in a primitive form with special concern for modelling interactions among web service instances in a network context.

2.1. Syntax

The *syntax* of WS-CALCULUS, given in Table 1, is parameterised with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: *properties* (sorts of late bound constants storing some relevant values within service instances, ranged over by p), *basic values* (integers `Int`, strings `Str`, and booleans) and corresponding *variables* (ranged over by b), *addresses* (ranged over by a) and *partner links* (namely variables storing addresses used to identify service partners within an interaction, ranged over by l), and *service identifiers* (ranged over by A) each with a fixed non-negative arity. The language is also parametric with respect to a set of *operations*, ranged over by o, which we do not specify, and *expressions*, ranged over by e, whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, basic values and variables, partner links, addresses and properties. Notationally, we will use u to range over *values* (i.e. basic values and addresses), v to range over *variables* (i.e. basic variables and partner links), w to range over *operation parameters* (i.e. variables and properties), c to range over *correlation patterns* (i.e. values and properties), and r to range over addresses and partner links. Addresses may be *underlined* to denote that they cannot be transmitted as operation parameters, while partner links may be subject to the operator $\ulcorner\_\urcorner$ that forces them to be already initialised.

Notation $\bar{\_}$ denotes tuples of objects. E.g. $\bar{v}$ is a tuple of variables; this will sometimes be written as $\bar{v}_{i\in I}$, for an appropriate index-set I, and v_i denotes the i-th element. We assume that variables in the same tuple are pairwise distinct. When convenient, we shall regard a tuple simply as a set writing e.g. $a \in \bar{u}$ to mean that a is an element of $\bar{u}$. All notations shall extend to tuples component-wise.

A WS-CALCULUS *node* can be thought of as a WS-BPEL service. Nodes, written as $a :: C$, are uniquely identified by an address a and a behavioural part C. Finite (non-empty) sets of nodes are called *networks*. Since we are interested in describing asynchronous interactions, we model each communication pattern by connecting one or more one-way operations. The more complex asynchronous request-response interaction pattern is expressed by connecting two one-way operations (request and callback). It is beneficial, for ex-

Table 1. WS-CALCULUS syntax

N	::=	$a :: C \mid N \cup N$	(networks)
C	::=	$*s \mid m \gg s \mid \langle a, o, \bar{u} \rangle \mid C \mid C$	(components)
m	::=	$\emptyset \mid \{p = u\} \mid m \cup m$	(correlation constraints)
s	::=		(services)
		$\mathbf{0}$	(empty)
	$\mid$	$\mathbf{exit}$	(exit)
	$\mid$	$\mathbf{ass}\,(\bar{w}, \bar{e})$	(assign)
	$\mid$	$\mathbf{inv}\,(r, o, \bar{w})$	(invoke)
	$\mid$	$\mathbf{rec}\,(r, o, \bar{w})$	(receive)
	$\mid$	$\mathbf{if}\ (e)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{s\}$	(conditional)
	$\mid$	$s; s$	(sequence)
	$\mid$	$s \mid s$	(flow)
	$\mid$	$\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$	(pick)
	$\mid$	$A(\bar{w})$	(call)

ample, to make web services asynchronous to reduce the time a requestor spends waiting for responses. By making a service call asynchronous, with a separate return message, the requestor will be able to continue execution while the provider has a chance to respond. This is not to say that synchronous service behavior is wrong, just that experience has demonstrated that asynchronous service behavior is desirable, especially when communication costs are high or network latency is unpredictable, and provides the developer with a simpler scalability model [BJK02]. Typically, to support an asynchronous mode of operation, interactions are developed as pairs of roles such that one role depends on another; for example, a client must implement a role to call a requestor because the client role provides some callback mechanism.

Components C may be service specifications, instances or requests. The behavioural specification of a service s is written $*s$, while $m \gg s'$ represents a service instance that behaves according to s' and whose properties evaluate according to the (possibly empty) set m of correlation constraints. A *correlation constraint* is a pair, written $p = u$, recording the value u assigned to the property p. Properties are used to store values that are important to identify service instances. For example, one might use a property named *purchase-order-id* to uniquely identify instances of a service that handles purchase orders. A service request $\langle a, o, \bar{u} \rangle$ represents an operation invocation that must still be processed and contains the invoker address a, the operation name o and the data $\bar{u}$ for operation execution. WS-CALCULUS operation names represent WS-BPEL pairs 'partner link – operation' (instead, WS-BPEL partner links are not explicitly modeled), thus the first two components of service requests represent an endpoint between interacting web services.

Services are structured activities built from *basic activities*, i.e. empty activity $\mathbf{0}$, instance forced termination $\mathbf{exit}$, assignment $\mathbf{ass}\,(\_, \_)$, service invocation $\mathbf{inv}\,(\_, \_, \_)$ and service request processing $\mathbf{rec}\,(\_, \_, \_)$, by exploiting operators for conditional choice $\mathbf{if}\ (\_)\ \mathbf{then}\ \{\_\}\ \mathbf{else}\ \{\_\}$, sequential composition $\_; \_$, parallel composition $\_ \mid \_$, external choice[1] $\sum_{i \in I} \mathbf{rec}\,(\_, \_, \_)\,; \_$ and service call $A(w_1, \cdots, w_n)$ where n is the arity of

[1] Whenever the external choice is between two activities s_1 and s_2, we shall simply write $s_1 + s_2$.

A. Every service identifier A with arity n has a unique definition of the form $A(\bar{v}_{i\in\{1,..,n\}}) \stackrel{def}{=} s$, where the v_i must be fresh and pairwise distinct.

The WS-CALCULUS binding constructs are $\mathbf{ass}\,(\bar{w},\bar{e})$ and $\mathbf{rec}\,(r,o,\bar{w})$ that bind the variables and the properties in $\bar{w}$. The latter also binds r if it is a partner link and is not subject to the operator $\ulcorner\_\urcorner$; in this case, we will say that r is *implicitly* initialised (otherwise, we will say that a partner link is *explicitly* initialised). In other words, $\ulcorner l \urcorner$ represents a free occurrence of l (e.g. a callback address) that must have been bound previously. For example, $\mathbf{rec}\,(\ulcorner l \urcorner, o, \bar{w})$ denotes a receive operation along the free partner link l. This way, we can use both free and bound occurrences of l as, for example, in the service $\mathbf{rec}\,(l,o',\bar{w}')\,;\mathbf{rec}\,(\ulcorner l \urcorner, o, \bar{w})$ to mean that the two receives expect messages coming from the same statically unknown address.

The scope of a binder extends to the whole component where the binder occurs (namely, like in WS-BPEL, variables and properties are global to the instance). A variable occurrence is free if it is not under the scope of a binder. Without loss of generality, we assume that all bound partner links are pairwise distinct, except for those occurring within alternative branches of conditional and pick constructs. Thus, the following fragment of service is well-defined:

$$\ldots\mathbf{if}\ (e)\ \mathbf{then}\ \{\ldots\mathbf{rec}\,(l,\ldots,\ldots)\ldots\}\ \mathbf{else}\ \{\ldots\mathbf{rec}\,(l,\ldots,\ldots)\ldots\};\mathbf{inv}\,(l,\ldots,\ldots)\ldots$$

In general, we use $fv(s)$ (resp. $bv(s)$) to denote the set of variables which occur free (resp. bound) in s. In particular, variables of $\bar{w}$ are free in $A(\bar{w})$. In a definition $A(\bar{v}) \stackrel{def}{=} s$ we assume $fv(s) \subseteq \bar{v}$.

In the sequel we shall only consider networks that are *well-formed* in the sense that they comply with the following syntactic constraints. Firstly, pairwise distinct nodes have different addresses. Secondly, if we call *start activities* of a service s all those activities that are not syntactically preceded by other ones, then at least one start activity of $*s$ must be a $\mathbf{rec}\,(\_,\_,\_)$ and, if multiple $\mathbf{rec}\,(\_,\_,\_)$ are enabled concurrently, then they must use the same non-empty set of properties.

2.2. Operational semantics

The *operational semantics* of WS-CALCULUS is given in terms of a structural congruence and of a reduction relation over networks. For simplicity, in this section we model taking place of errors (e.g. when the premises of all reduction rules are not satisfied) simply as deadlock.

The semantics of networks will be defined over an enriched set of nets that also includes those auxiliary nets resulting from replacing (free occurrences of) variables with values in nets produced by the syntax of Table 1. Therefore, we will let free occurrences of v (and w) to also denote corresponding values, and a to also denote possibly underlined addresses.

The *structural congruence*, denoted by $\equiv$ and defined as the smallest congruence relation induced by the rules in Table 2, identifies syntactically different terms which intuitively represent the same term. In Table 2, and in the sequel, notation $\_s$ shall denote both service specifications ($*s$) and service instances ($m \gg s$). At the level of services, the structural congruence states that: services only differing for the bound variables are the same (*alpha-conversion*); the sequence operator is associative and has $\mathbf{0}$ as identity element

Table 2. Structural congruence $\equiv$

$$\frac{s \text{ and } s' \ \alpha\text{-equivalent}}{s \equiv s'}$$

$$\begin{array}{rcl} s;\mathbf{0} & \equiv & s \\ \mathbf{0};s & \equiv & s \\ (s;s');s'' & \equiv & s;(s';s'') \end{array} \qquad \begin{array}{rcl} s \mid \mathbf{0} & \equiv & s \\ s \mid s' & \equiv & s' \mid s \\ s \mid (s' \mid s'') & \equiv & (s \mid s') \mid s'' \end{array} \qquad \begin{array}{rcl} s + \mathbf{0} & \equiv & s \\ s + s' & \equiv & s' + s \\ s + (s' + s'') & \equiv & (s + s') + s'' \end{array}$$

$$\frac{s \equiv s'}{\_s \mid C \equiv \_s' \mid C} \qquad \begin{array}{rcl} C \mid m \gg \mathbf{0} & \equiv & C \\ C_1 \mid C_2 & \equiv & C_2 \mid C_1 \\ (C_1 \mid C_2) \mid C_3 & \equiv & C_1 \mid (C_2 \mid C_3) \end{array} \qquad \frac{C \equiv C'}{\{a :: C\} \equiv \{a :: C'\}} \qquad \frac{N_1 \equiv N_1'}{N_1 \cup N_2 \equiv N_1' \cup N_2}$$

(law $\mathbf{0};s \equiv s$ is exploited to enable a new activity when a preceding one terminates); the flow and the pick operators are commutative, associative and have $\mathbf{0}$ as identity element. The remaining rules of Table 2 extend the structural congruence to components and networks, and should be self-explicative. In particular, components composition is commutative and associative, and has $m \gg \mathbf{0}$ as identity element (i.e. instances of this form are terminated instances and, thus, can be removed).

The *reduction relation* over networks, written $\succ\!\!\longrightarrow$, relies on a labelled transition relation $\xrightarrow{\alpha}$ over service instances, where label α is generated by the following productions:

$$\alpha \ ::= \ \phi_{exit} \ \mid \ \bar{w} := \bar{u} \ \mid \ \langle a, o, \bar{u} \rangle \ \mid \ ?(r, o, \bar{w})$$

The meaning of labels is as follows: ϕ_{exit} denotes forced termination of a service instance, $\bar{w} := \bar{u}$ denotes execution of a multiple assignment, $\langle a, o, \bar{u} \rangle$ denotes invocation of operation o located at a with data $\bar{u}$, and $?(r, o, \bar{w})$ denotes providing the operation o with parameters $\bar{w}$ on request by a service instance located at r.

To define $\xrightarrow{\alpha}$ we need a function for evaluating expressions: it takes an expression and returns a basic value or an address. We write $m \rhd e$ such a function, but we do not explicitly define it because the exact syntax of expressions is deliberately not specified (recall that WS-CALCULUS is parameterised wrt the syntax of expressions). Expressions to be evaluated can contain properties; thus, evaluation of e takes place wrt a set of correlation constraints m storing the values of the properties that may occur within e. On the contrary, expressions to be evaluated cannot contain (free) variables because these occurrences are replaced with the corresponding values as soon as the variables are bound. Indeed, execution of a binding construct generates a *substitution* σ, i.e. a map from basic variables to basic values and from partner links to addresses, that is applied to the whole instance where the binder occurs. A substitution σ will be sometimes written as $(\bar{v} \mapsto \sigma(\bar{v}))$ for $\bar{v} = dom(\sigma)$. Application of substitution σ to s is written $s \cdot \sigma$. The effect of $s \cdot \sigma$ is that, for each $x \in dom(\sigma)$, every free occurrence of x in s is replaced with $\sigma(x)$. We use $\sigma_1 \circ \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

Now, $\xrightarrow{\alpha}$ can be defined as the least relation over service instances induced by the rules in Table 3. For simplicity, we explicitly write only those entities that are necessary for a transition to occur or are modified by it. For example, since correlation constraints are sometimes necessary but are never modified by a transition, we write the relation as $m \vdash s \xrightarrow{\alpha} s'$ instead of $m \gg s \xrightarrow{\alpha} m \gg s'$. The most of the rules are straightforward, we only remark a few points. Rule (*Receive*) states that a $\mathbf{rec}\,(\_, \_, \_)$ cannot be performed

Table 3. WS-CALCULUS operational semantics: instances

$$m \vdash \mathbf{exit} \xrightarrow{\phi_{exit}} \mathbf{0} \quad (Exit)$$

$$m \vdash \mathbf{ass}\,(\bar{w}, \bar{e}) \xrightarrow{\bar{w} := (m \rhd \bar{e})} \mathbf{0} \quad (Assign)$$

$$m \vdash \mathbf{inv}\,(a, o, \bar{c}) \xrightarrow{\langle a, o, m \rhd \bar{c} \rangle} \mathbf{0} \quad (Invoke)$$

$$\frac{r \neq \ulcorner l \urcorner}{m \vdash \mathbf{rec}\,(r, o, \bar{w}) \xrightarrow{?(r,o,\bar{w})} \mathbf{0}} \quad (Receive)$$

$$\frac{(m \rhd e) = \mathbf{true} \qquad m \vdash s_1 \xrightarrow{\alpha} s_1'}{m \vdash \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} \xrightarrow{\alpha} s_1'} \quad (If_{true})$$

$$\frac{(m \rhd e) = \mathbf{false} \qquad m \vdash s_2 \xrightarrow{\alpha} s_2'}{m \vdash \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} \xrightarrow{\alpha} s_2'} \quad (If_{false})$$

$$\frac{m \vdash s_1 \xrightarrow{\alpha} s_1'}{m \vdash s_1; s_2 \xrightarrow{\alpha} s_1'; s_2} \quad (Sequence)$$

$$\frac{m \vdash s_1 \xrightarrow{\alpha} s_1'}{m \vdash s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \quad (Flow)$$

$$\frac{m \vdash \mathbf{rec}\,(r, o, \bar{w})\,; s \xrightarrow{?(r,o,\bar{w})} s'}{m \vdash \mathbf{rec}\,(r, o, \bar{w})\,; s + \sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i \xrightarrow{?(r,o,\bar{w})} s'} \quad (Pick)$$

$$\frac{m \vdash s \cdot (\bar{v} \mapsto (m \rhd \bar{c})) \xrightarrow{\alpha} s'}{m \vdash A(\bar{c}) \xrightarrow{\alpha} s'}\ A(\bar{v}) \stackrel{def}{=} s \quad (Call)$$

when the address of the sender of a request is unknown and cannot be learned (i.e. the first argument is neither an address nor a link). Notice that if $r = \ulcorner l \urcorner$ the operation cannot proceed since the partner link used in the receive is still subject to some binder and thus practically unknown. Rule (*Flow*) states that executions of the two argument services are interleaved. Rule (*Pick*) states that the pick activity can execute any of its receive activities and then proceed accordingly.

To define the reduction relation, we need a mechanism for checking if the assignments of u_i to w_i, for any index i in a given set I, comply with the correlation constraints in a given m. Such a mechanism is rendered as a function, written $m \rhd (\bar{w}_{i \in I} := \bar{u}_{i \in I})$, that in case the check succeeds, returns a pair $\langle m', \sigma \rangle$ where m' is the set of the correlation constraints for the properties in $\bar{w}_{i \in I}$ and σ is the substitution for the variables in $\bar{w}_{i \in I}$. It is defined inductively on $\bar{w}$ as follows:

$m \rhd (v := u) = \langle \emptyset, (v \mapsto u) \rangle$

$m \rhd (a := a) = \langle \emptyset, \emptyset \rangle$

$$m \rhd (p := u) = \begin{cases} \langle \emptyset, \emptyset \rangle & \text{if } p = u \in m \\ \langle \{p = u\}, \emptyset \rangle & \text{if } \nexists\, u' \text{ s.t. } p = u' \in m \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$m \rhd (w, \bar{w} := u, \bar{u}) = \langle m' \cup m'', \sigma \circ \sigma' \rangle$ if $m \rhd (w := u) = \langle m', \sigma \rangle$ and $m \cup m' \rhd (\bar{w} := \bar{u}) = \langle m'', \sigma' \rangle$

We also exploit the following two functions:

- $P(\bar{w})$ returns the set of properties contained in $\bar{w}$ and is defined as follows:
  $P(v) = P(u) = \emptyset$
  $P(p) = \{p\}$
  $P(\langle w_1, \ldots, w_n \rangle) = P(w_1) \ \cup \ \ldots \ \cup \ P(w_n)$
- $R(C, m, \langle a, o, \bar{u} \rangle)$ returns the multiset of activities of the form $\mathbf{rec}\,(\_, \_, \_)$ that are start activities of C, match the message $\langle a, o, \bar{u} \rangle$, and comply with the constraints m. It is defined inductively on the syntax

Table 4. WS-CALCULUS operational semantics: networks

$$\frac{m \vdash s \xrightarrow{\bar{w}:=\bar{u}} s' \qquad m \rhd (\bar{w} := \bar{u}) = \langle m', \sigma \rangle}{\{a :: m \gg s \mid C\} \rightarrowtail \{a :: (m \cup m') \gg s' \cdot \sigma \mid C\}} \ (Assign_N)$$

$$\frac{m \vdash s \xrightarrow{\langle a_2, o, \bar{u} \rangle} s' \qquad \underline{a} \notin \bar{u}}{\{a_1 :: m \gg s \mid C_1,\ a_2 :: C_2\} \rightarrowtail \{a_1 :: m \gg s' \mid C_1,\ a_2 :: \langle a_1, o, \bar{u} \rangle \mid C_2\}} \ (Invoke_N)$$

$$\frac{m \vdash s \xrightarrow{?(r,o,\bar{w})} s' \qquad m \rhd (r, \bar{w} := \underline{a}', \bar{u}) = \langle m', \sigma \rangle \qquad \#R(s, m, \langle a', o, \bar{u} \rangle) = 1}{\{a :: m \gg s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: (m \cup m') \gg s' \cdot \sigma \mid C\}} \ (Receive_I)$$

$$\frac{\emptyset \vdash s \xrightarrow{?(r,o,\bar{w})} s' \qquad \emptyset \rhd (r, \bar{w} := \underline{a}', \bar{u}) = \langle m, \sigma \rangle \qquad \#R(*s \mid C, \_, \langle a', o, \bar{u} \rangle) = 1}{\{a :: *s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: *s \mid m \gg s' \cdot \sigma \mid C\}} \ (Receive_S)$$

$$\frac{m \vdash s \xrightarrow{\phi_{exit}} s'}{\{a :: m \gg s \mid C\} \rightarrowtail \{a :: C\}} \ (Terminate) \qquad \frac{N_1 \rightarrowtail N_1'}{N_1 \cup N_2 \rightarrowtail N_1' \cup N_2} \ (Part) \qquad \frac{N \equiv N_1 \quad N_1 \rightarrowtail N_2 \quad N_2 \equiv N'}{N \rightarrowtail N'} \ (Cong)$$

of C as follows:

$$R(*s, \_, \langle a, o, \bar{u} \rangle) = R(s, \emptyset, \langle a, o, \bar{u} \rangle) \qquad R(m \gg s, \_, \langle a, o, \bar{u} \rangle) = R(s, m, \langle a, o, \bar{u} \rangle)$$

$$R(\langle a, o, \bar{u} \rangle, \_, \_) = \emptyset \qquad R(C_1 \mid C_2, \_, \langle a, o, \bar{u} \rangle) = R(C_1, \_, \langle a, o, \bar{u} \rangle) \uplus R(C_2, \_, \langle a, o, \bar{u} \rangle)$$

$$R(\mathbf{0}, \_, \_) = R(\mathbf{exit}, \_, \_) = R(\mathbf{inv}\,(r, o, \bar{w})\,, \_, \_) = R(\mathbf{ass}\,(\bar{v}, \bar{e})\,, \_, \_) = \emptyset$$

$$R(\mathbf{rec}\,(r, o, \bar{w})\,, m, \langle a, o, \bar{u} \rangle) = \{\mathbf{rec}\,(r, o, \bar{w})\} \quad \text{if } m \rhd (r, \bar{w} := a, \bar{u}) \neq \mathit{undef}$$

$$R(\mathbf{if}\,(e)\,\mathbf{then}\,\{s_1\}\,\mathbf{else}\,\{s_2\}, m, \langle a, o, \bar{u} \rangle) = R(s_1 \mid s_2, m, \langle a, o, \bar{u} \rangle) = R(s_1, m, \langle a, o, \bar{u} \rangle) \uplus R(s_2, m, \langle a, o, \bar{u} \rangle)$$

$$R(s_1; s_2, m, \langle a, o, \bar{u} \rangle) = R(s_1, m, \langle a, o, \bar{u} \rangle)$$

$$R(\textstyle\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i, m, \langle a, o, \bar{u} \rangle) = \biguplus_{i \in I} R(\mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,, m, \langle a, o, \bar{u} \rangle)$$

$$R(A(\bar{c}), m, \langle a, o, \bar{u} \rangle) = R(s \cdot (\bar{v} \mapsto (m \rhd \bar{c})), m, \langle a, o, \bar{u} \rangle) \quad \text{if } A(\bar{v}) \stackrel{def}{=} s$$

In the sequel, we will use $\#R(C, m, \langle a, o, \bar{u} \rangle)$ to indicate the cardinality of the multiset returned by $R(C, m, \langle a, o, \bar{u} \rangle)$.

Finally, we can define the reduction relation $\rightarrowtail$ as the least relation over networks induced by the rules in Table 4. Let us now comment on the rules. Rule ($Assign_N$) states that the effect of an assignment is global wrt the instance and consists of replacing the free occurrences of the variables bound by the assignment with the corresponding values and of extending the set of correlation constraints identifying the instance with the pairs resulting from the assignment. Rule ($Invoke_N$) states that service invocation corresponds to adding a service request to the dataspace of the invoked service provided that no address implicitly received (see rules ($Receive_I$) and ($Receive_S$)) is exported as operation parameter. The request is a tuple, containing the address a_1 of the invoker, the name of the invoked operation o and the message $\bar{u}$ (i.e. the arguments to be passed to o). Hence, the invocation of a remote service is asynchronous because the invoker can proceed before its request is processed. WS-BPEL also provides a synchronous invocation that forces

the invoker to wait for an answer by the invoked service, which indeed performs a pair of activities *receive – reply*. In WS-CALCULUS, this behaviour is rendered through a pair of activities *invoke – receive* executed by the invoker and a pair of activities *receive – invoke* executed by the invoked service. Rule ($Receive_I$) states that activity receive cannot progress until a matching request has been received. Thus, differently from activity invoke, it is blocking. Requests are delivered to the correct service instance by exploiting the partner link and the operation contained in the request, which must coincide with those in the label of the transition performed by the service instance, and by exploiting the correlation constraints identifying the specific instance, which must enable the assignment of the values contained in the request to the parameters contained in the receive. In case the address of the invoker is not known in advance, i.e. $r = l$, after the reduction, the address contained in the request is marked as not further transmissible and used to replace the partner link occurring in the receive. When the reduction takes place, the matching request is consumed and the effect on the instance is the same as that of the corresponding assignment. If $\#R(s, m, \langle a', o, \bar{u}\rangle) \neq 1$, i.e. in s there are two or more receive activities simultaneously enabled for the same combination of operation and correlation set to consume the same message, then these receive conflicts prevent communication to take place. The last rule for the activity receive, ($Receive_S$), permits to create a new service instance on receipt of a request that cannot be delivered to an existing instance. The premise $\#R(*s \mid C, \_, \langle a', o, \bar{u}\rangle) = 1$ prevents interferences with the first rule for receive in case of multiple start activities, as illustrated by the example

$$\{a :: *(\,(\mathbf{rec}\,(l, o, \langle p\rangle) \mid \mathbf{rec}\,(l', o', \langle p\rangle)\,)\,; s) \quad | \quad \{p = 10\} \gg \mathbf{rec}\,(l'', o, \langle p\rangle)\,; s \quad | \quad \langle a', o, \langle 10\rangle\rangle\}$$

where only the service instance can evolve. Rule (*Terminate*) states that the whole service instance performing a transition labelled ϕ_{exit} immediately terminates. Rule (*Part*) states that if a part of a larger network evolves, the whole network evolves accordingly. Rule (*Cong*) is standard and states that structural congruent networks have the same reductions.

3. From WS-BPEL to ws-calculus

WS-BPEL is essentially a linguistic layer on top of WSDL (*Web Services Description Language*, [CCMW01]) for describing the structural aspects of web service orchestrations. In WS-BPEL, the logic of interaction between a service and its environment is described in terms of structured patterns of communication actions composed by means of control-flow constructs that enable the representation of complex structures. The orchestration exploits state information that is maintained through shared variables and managed through message correlation. For the specification of orchestration, WS-BPEL provides many different activities that are distinguished between *basic activities* and *structured activities*.

The following basic activities are provided: `<receive>` and `<reply>`, to provide web service one-way and request-response operations; `<invoke>`, to invoke web service operations; `<wait>`, to delay execution for some amount of time; `<assign>`, to update variables with new data; `<throw>`, to signal internal faults; `<exit>`, to immediately end the service instance; `<empty>`, to do nothing; `<compensate>` and `<compensateScope>`,

to invoke compensation handlers; `<rethrow>` to propagate faults; `<validate>`, to validate variables; and `<extensionActivity>`, to add new activity types.

The structured activities describe the control-flow logic of a business process by composing basic and/or structured activities recursively. The following structured activities are provided: `<sequence>`, to process activities sequentially; `<if>`, to process activities conditionally; `<while>` and `<repeatUntil>`, to repetitively execute activities; `<flow>`, to process activities in parallel; `<pick>`, to process activities selectively; `<forEach>`, to (sequentially or in parallel) process multiple activities; and `<scope>`, to add some handlers for the exception signals that may arise during the execution of a primary activity.

Notably, synchronization dependencies among activities, other than by means of control-flow constructs, can also be specified through *control links* to form directed acyclic graphs, called flow graphs. A control link is a conditional transition that connects a 'source' activity to a 'target' activity. When a source activity completes, the associated *transition condition* is evaluated to determine the status of the *join condition* of effect on the control link on the target activity. A target activity may only start when all its source activities complete and its join condition has been evaluated to true.

The handlers within a `<scope>` can be of four different kinds: `<faultHandler>`, to provide the activities in response to faults occurring during a scope execution; `<compensationHandler>`, to provide the activities to compensate successful executed scopes; `<terminationHandler>`, to control the forced termination of a scope; and `<eventHandler>`, to process message or timeout events. If a fault occurs during execution of a scope, the control is transferred to the corresponding fault handler and all currently running activities of the scope are interrupted immediately without involving any fault/compensation handling behaviour. If another fault occurs during a fault/compensation handling, then it is re-thrown, possibly, to the immediately enclosing scope. Compensation handlers attempt to reverse the effects of previously successfully completed scopes and have been introduced to support Long-Running (Business) Transactions (LRTs). Compensation can only be invoked from within fault or compensation handlers starting the compensation either of a specific inner (completed) scope, or of all inner completed scopes in the reverse order of completion. The latter alternative is also called the *default* compensation behaviour. Invoking a compensation handler that is unavailable is equivalent to perform an empty activity.

Finally, a WS-BPEL program is a business `<process>`, that is a sort of `<scope>` without compensation and termination handlers.

4. Example: A shipping service

We consider an extended version of the shipping service described in the official specification of WS-BPEL [OAS07] (Section 15.1). This example will allow us to illustrate most of the language features, including correlation sets, shared variables and flow control structures.

The shipping service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The service offers two types of shipment: shipments where the items are held and

shipped together and shipments where the items are shipped piecemeal until the order is fulfilled. An abstract service description, that does not include specific details on, e.g., the handling of error conditions, is as follows:

```
receive shipOrder
if
  condition shipComplete
    send shipNotice
  else
    itemsShipped := 0
    while itemsShipped < itemsTotal
      itemsCount := opaque   // non-deterministic assignment corresponding e.g. to
                             // internal interaction with back-end system
      send shipNotice
      itemsShipped = itemsShipped + itemsCount
```

As reported in [OAS07], the corresponding WS-BPEL program is as follows (basic activities receive/invoke and assign are highlighted to make the reading of the code easier):

```
<process name="shippingService">
...
<correlationSets>
  <correlationSet name="shipOrder" properties="props:shipOrderID" />
</correlationSets>
<sequence>
<receive partnerLink="customer" operation="shippingRequest"
  variable="shipRequest">
  <correlations>
    <correlation set="shipOrder" initiate="yes" />
  </correlations>
</receive>
<if>
  <condition>
    bpel:getVariableProperty('shipRequest','props:shipComplete')
  </condition>
  <sequence>
    <assign>
      <copy>
        <from variable="shipRequest" property="props:shipOrderID" />
        <to variable="shipNotice" property="props:shipOrderID" />
      </copy>
      <copy>
        <from variable="shipRequest" property="props:itemsCount" />
        <to variable="shipNotice" property="props:itemsCount" />
      </copy>
    </assign>
    <invoke partnerLink="customer" operation="shippingNotice"
      inputVariable="shipNotice">
      <correlations>
        <correlation set="shipOrder" pattern="request" />
      </correlations>
    </invoke>
  </sequence>
  <else>
    <sequence>
      <assign>
        <copy>
          <from>0</from>
          <to>$itemsShipped</to>
        </copy>
      </assign>
```

Table 5. Mapping of the Shipping service

Element	WS-BPEL	WS-CALCULUS
correlation property	`shipOrderID`	$p\_orderId$
partner link	`customer`	$cust$
operation	`shippingRequest`	$shipReq$
variable	`shipRequest`	$\langle p\_orderId, complete, itemsTot\rangle$
operation	`shippingNotice`	$shipNot$
variable	`shipNotice`	$\langle p\_orderId, itemsCount\rangle$
variable	`itemsShipped`	$itemsShipped$

```
          <while>
            <condition>
              $itemsShipped
              &lt;
              bpel:getVariableProperty('shipRequest','props:itemsTotal')
            </condition>
            <sequence>
              <assign>
                <copy>
                  <opaqueFrom />
                  <to variable="shipNotice" property="props:shipOrderID" />
                </copy>
                <copy>
                  <opaqueFrom />
                  <to variable="shipNotice" property="props:itemsCount" />
                </copy>
              </assign>
              <invoke partnerLink="customer" operation="shippingNotice"
                inputVariable="shipNotice">
                <correlations>
                  <correlation set="shipOrder" pattern="request" />
                </correlations>
              </invoke>
              <assign>
                <copy>
                  <from>
                    $itemsShipped
                    +
                    bpel:getVariableProperty('shipNotice','props:itemsCount')
                  </from>
                  <to>$itemsShipped</to>
                </copy>
              </assign>
            </sequence>
          </while>
        </sequence>
      </else>
    </if>
    </sequence>
    </process>
```

The definition of the shipping service exploits a receive activity to receive orders from clients, an assign activity to handle the received data, and an invoke activity to send shipping notices to clients (see the mapping in Table 5). An order request contains an order identifier that is used to correlate the shipping notice(s) with the shipping order. The flow control of the service is defined by means of conditional and iterative constructs (modelled through recursive definitions). Finally, the whole service is modelled by the node $a :: *s$ where the service specification is as follows:

$$
\begin{array}{rcl}
s & \triangleq & \mathbf{rec}\,(cust, shipReq, \langle p\_orderId, complete, itemsTot\rangle)\,; \\
 & & \mathbf{if}\ (complete)\ \mathbf{then}\ \{ \\
 & & \quad \mathbf{ass}\,(itemsCount, itemsTot)\,; \\
 & & \quad \mathbf{inv}\,(cust, shipNot, \langle p\_orderId, itemsCount\rangle) \\
 & & \} \\
 & & \mathbf{else}\ \{ \\
 & & \quad \mathbf{ass}\,(itemsShipped, 0)\,; \\
 & & \quad B(\langle itemsShipped, itemsTot, p\_orderId, cust\rangle) \\
 & & \}
\end{array}
$$

$$
\begin{array}{rcl}
B(\langle items, tot, ordId, l_c\rangle) & \stackrel{def}{=} & \mathbf{ass}\,(itemC, rand(tot - items))\,; \\
 & & \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC\rangle)\,; \\
 & & \mathbf{ass}\,(itemsShipped', items + itemC)\,; \\
 & & \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then} \\
 & & \{B(\langle itemsShipped', tot, ordId, l_c\rangle)\}\ \mathbf{else}\ \{\mathbf{0}\}
\end{array}
$$

where:

- *cust* is the partner link used by the service to communicate with its clients;
- *shipReq* and *shipNot* are the operations used to receive the shipping request and to send shipping notices, respectively;
- $\langle p\_orderId, complete, itemsTot\rangle$ is the tuple of parameters used for the request shipping message; each instance of the shipping service is uniquely identified by the correlation set composed by the property $p\_orderId$;
- $\langle p\_orderId, itemsCount\rangle$ is the tuple of parameters used for the response message; we notice that the property $p\_orderId$ is used to fill the first field of the response message;
- *itemsShipped* is an integer variable used as a counter;
- $\langle items, tot, ordId, l_c\rangle$ is the tuple of the formal parameters of the definition B;
- $rand(k)$ is a function which returns a random integer number not greater than k; it represents an internal interaction with the back-end system.

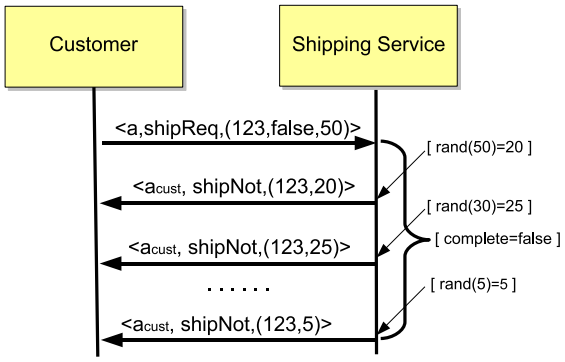
Now, consider the following composition of the shipping service definition containing a customers invocation of the service

$$a :: *s \mid \langle a_{cust}, shipReq, \langle 123, \mathbf{false}, 50\rangle\rangle$$

In the first computational step, the customer's invocation is consumed and an instance of the shipping service is created. Thus the overall computation becomes

$$a :: *s \mid \{p\_orderId = 123\} \gg \mathbf{ass}\,(itemsShipped, 0)\,; B(\langle itemsShipped, 50, p\_orderId, a_{cust}\rangle)$$

Fig. 1. Graphical representation of a shipping service scenario



The computation can now go on, e.g., as shown in the customized UML sequence diagram of Figure 1, where it is supposed that the inner scope the shipping processing successfully completes while its continuation terminate, e.g., the condition $itemsShipped < itemsTot$ holds false.

5. Extensions of ws-calculus

In this section, we present some features not initially included in our calculus and, in particular, we extend WS-CALCULUS in order to model flow graphs, timed activities, scopes and compensation handling.

5.1. Flow graphs

A well-known characteristic of WS-BPEL is that its set of structured activities is not intended to be the minimal required set. There are cases, for example, where one activity can replace another. This is the case of the *flow activity*, used in [OAS07] to structure workflow processing, that may be easily encoded within WS-CALCULUS. Here, we present a possible encoding of the flow graph activity in WS-CALCULUS.

In WS-BPEL, parallel execution of activities may be synchronized by establishing synchronization dependencies (here called *flow links* or *links* for short) among activities. At the beginning of the parallel execution, all links are inactive and only those activities with no synchronization dependencies can execute. When the execution of an activity completes, a *transition condition* is evaluated to determine the status of the outgoing links that can be *positive*, *negative* or *undefined*. Once all incoming links to an activity are active (that is, they have been assigned either a positive or negative state), a guard, called *join condition*, is evaluated. When an activity in the flow graph cannot execute (that is, the join condition fails), a *join failure* fault can be emitted to signal that some activities in the flow are not completed. An attribute called *suppress join failure* can be set to *yes* to ensure that join condition failures do not cause the flow activity to throw the *join failure* fault.

Default values for join and transition conditions are the logical 'OR' of the status of the incoming links and the boolean value **true** respectively, while the default setting of the *suppress join failure* attribute is *no*.

In order to introduce the flow graph activity, we extend the syntax of WS-CALCULUS as illustrated in Table

Table 6. WS-CALCULUS plus Flow links

s	$::=$	$\ldots \mid ls \mid_L ls$	(flow graph)
ls	$::=$	$(jc) \stackrel{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})$	(join condition and outgoing links)
	$\mid$	$s \Rightarrow (\overline{fl}, \bar{e})$	(outgoing links)
jc	$::=$	$\mathbf{true} \mid \mathbf{false} \mid fl \mid \neg jc \mid jc \vee jc \mid jc \wedge jc$	(join conditions)
sjf	$::=$	$yes \mid no$	(suppress join failure)

6. A flow graph activity $ls \mid_L ls$ is the parallel composition of two "linked" services ls, which can synchronize by means of the set of flow links denoted by L. Here, we assume that different flow link sets used by a service specification must be pairwise disjoint. A linked service ls is equipped with a set of incoming flow links that forms the join condition, and a set of outgoing flow links that represent the transition conditions. We denote incoming flow links and join condition with $(jc) \stackrel{sjf}{\Rightarrow}$. The outgoing links are represented by $\Rightarrow (\overline{fl}_{i \in I}, \bar{e}_{i \in I})$ where each pair (fl_i, e_i) denotes a transition condition, such that fl_i is a flow link and e_i is a boolean expression. The attribute sjf permits to suppress a possible join failure, obtaining the so called *Dead-Path Elimination* (DPE) effect (see [OAS07]).

The encoding [2] of flow graphs in WS-CALCULUS is shown in Table 7. The auxiliary function $outLinkOf(s)$ returns the tuple (the order is not important) of the outgoing links used by s and is defined as follows:

$$\left.\begin{array}{r} outLinkOf(s \Rightarrow (\overline{fl}, \bar{e})) \\ outLinkOf((jc) \stackrel{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})) \end{array}\right\} = outLinkOf(s)\,,\, \overline{fl} \qquad \left.\begin{array}{r} outLinkOf(\mathbf{0}) \\ outLinkOf(\mathbf{exit}) \\ outLinkOf(\mathbf{ass}\,(\bar{w}, \bar{e})) \\ outLinkOf(\mathbf{inv}\,(r, o, \bar{w})) \\ outLinkOf(\mathbf{rec}\,(r, o, \bar{w})) \end{array}\right\} = \emptyset$$

$$outLinkOf(A(\bar{x})) = outLinkOf(s) \quad \text{if } A(\bar{v}) \stackrel{def}{=} s$$

$$\left.\begin{array}{r} outLinkOf(\mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\}) \\ outLinkOf(s_1; s_2) \\ outLinkOf(s_1 \mid s_2) \end{array}\right\} = outLinkOf(s_1)\,,\, outLinkOf(s_2)$$

$$outLinkOf(\textstyle\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i) = \cup_{i \in I}\, outLinkOf(s_i)$$

$$outLinkOf(ls_1 \mid_L ls_2) = outLinkOf(ls_1)\,,\, outLinkOf(ls_2)$$

In the following, we comment some interesting encoding rules. Primarily, we say that a flow graph processing is considered as a usual parallel composition. A join condition is encoded as boolean condition within an **if-then-else** statement, in which the transition conditions are translated by means of assignments of forms $\mathbf{ass}\,(\overline{fl}, e)$. In case of *suppress join failure* equals to *no*, a join condition failure produces a fault signal (by means of the activity $\mathbf{throw}\,(\phi_{joinfault})$) that can be caught by a proper fault handler (for this feature, we refer the interested readers to Section 5.3 where fault handlers are introduced). Structured pick and conditional activities are translated so that, when an activity is selected, the outgoing links from the other activities of discarded branch are set to $false$.

[2] We have deliberately omitted the uninteresting cases, such as **rec**, **inv**, etc.

Table 7. Flow link encoding

$$
\begin{array}{rcl}
[\![ls \mid_L ls]\!] & = & [\![ls]\!] \mid [\![ls]\!] \\
[\![(jc) \overset{y \in s}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & \mathbf{if}\ (jc)\ \mathbf{then}\ \{[\![s]\!]; \mathbf{ass}\left(\overline{fl}, \bar{e}\right)\}\ \mathbf{else}\ \{\mathbf{ass}(outLinkOf(s), \overline{\mathbf{false}})\} \\
[\![(jc) \overset{no}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & \mathbf{if}\ (jc)\ \mathbf{then}\ \{[\![s]\!]; \mathbf{ass}\left(\overline{fl}, \bar{e}\right)\}\ \mathbf{else}\ \{\mathbf{throw}\ (\phi_{joinfault})\} \\
[\![s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & [\![s]\!]; \mathbf{ass}(\overline{fl}, \bar{e}) \\
[\![\mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\}]\!] & = & \mathbf{if}\ (e)\ \mathbf{then}\ \{\mathbf{ass}(outLinkOf(s_2), \overline{\mathbf{false}}); [\![s_1]\!]\} \\
 & & \qquad \mathbf{else}\ \{\mathbf{ass}(outLinkOf(s_1), \overline{\mathbf{false}}); [\![s_2]\!]\} \\
[\![\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i]\!] & = & \sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; \mathbf{ass}(\bigcup_{j \in I\,, j \neq i} outLinkOf(s_j), \overline{\mathbf{false}}); [\![s_i]\!]
\end{array}
$$

Table 8. Timed WS-CALCULUS

n	::=	$a ::_t C$	(nodes)
C	::=	$\ldots \mid m, t \gg s$	(timed instances)
s	::=	$\ldots \mid \sum_{i \in I} g_i; s_i \mid \mathbf{wait}_{\,\mathrm{f}}(\delta) \mid \mathbf{wait}_{\,\mathrm{u}}(t)$	(wait activities)
g	::=	$\mathbf{rec}\,(r, o, \bar{w}) \mid \mathbf{wait}_{\,\mathrm{f}}(\delta) \mid \mathbf{wait}_{\,\mathrm{u}}(t)$	(guards)

5.2. Timed activities

A timed extension of WS-CALCULUS is presented in Table 8 in order to model *waiting* activities of WS-BPEL that allow to wait for a given time period or until a certain time has passed.

The syntax of WS-CALCULUS is now parameterized with respect to a new set `Time` of *time units* used to represent absolute time units (ranged over by t, t', ...) and time slots (ranged over by δ, δ',...). Thus, the syntax is extended so that each node of a network is equipped with a clock t, representing an absolute time for each located service instance[3]. Two timed activities $\mathbf{wait}_{\,\mathrm{f}}(t)$ and $\mathbf{wait}_{\,\mathrm{u}}(t)$ permit to specify, respectively, the delay for a certain period of time or until a certain deadline is reached. Consequently, the pick activity is modified in order to wait occurrences for a message arrive or for a time-out event.

Extended Semantics. The extended semantics is given in terms of transitions between configurations $m, t \gg s \xrightarrow{\alpha} m, t' \gg s'$, where t and t' are time units and the label α is now generated by

$$\alpha \ ::= \ \ldots \ \mid \ \delta$$

This kind of labels permits to express events corresponding to time elapsing. Some relevant transition rules are presented in Table 9. Given a time slot δ and a clock t, the auxiliary function $timeOut(g, \delta, t)$ returns a boolean value stating that the time period has expired for the specified guard g:

$$timeOut(g, \delta, t) = \begin{cases} \mathbf{true} & \text{if } g = \mathbf{wait}_{\,\mathrm{f}}(\delta') \text{ with } \delta' \leq \delta \\ \mathbf{true} & \text{if } g = \mathbf{wait}_{\,\mathrm{u}}(t + \delta) \text{ with } \delta' \leq t + \delta \\ \mathbf{false} & otherwise \end{cases}$$

Evaluations of expressions and basic activities, except for $\mathbf{wait}_{\,\mathrm{f}}(t)$ and $\mathbf{wait}_{\,\mathrm{u}}(t)$, have an instantaneous

[3] For each new service instantiation, the clock t is passed to the new instance in order to properly synchronize its local clock.

Table 9. Timed WS-CALCULUS operational semantics: instances

$$m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta} m, t+\delta \gg \mathbf{0} \quad (Wait_{\mathrm{f}}1) \qquad \frac{\delta' < \delta}{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta'} m, t+\delta' \gg \mathbf{wait}_{\mathrm{f}}(\delta - \delta')} \quad (Wait_{\mathrm{f}}2)$$

$$\frac{t+\delta = t'}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t') \xrightarrow{\delta} m, t' \gg \mathbf{0}} \quad (Wait_{\mathrm{u}}1) \qquad \frac{t+\delta < t'}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t') \xrightarrow{\delta} m, t+\delta \gg \mathbf{wait}_{\mathrm{u}}(t')} \quad (Wait_{\mathrm{u}}2)$$

$$m, t \gg \mathbf{rec}(r, o, \bar{w}) \xrightarrow{\delta} m, t+\delta \gg \mathbf{rec}(r, o, \bar{w}) \quad (tReceive) \qquad \frac{m, t \gg s_1 \xrightarrow{\delta} m, t+\delta \gg s_1'}{m, t \gg s_1; s_2 \xrightarrow{\delta} m, t+\delta \gg s_1'; s_2} \quad (tSequence)$$

$$\frac{m, t \gg s_1 \xrightarrow{\delta} m, t+\delta \gg s_1' \quad m, t \gg s_2 \xrightarrow{\delta} m, t+\delta \gg s_2'}{m, t \gg s_1 \mid s_2 \xrightarrow{\delta} m, t+\delta \gg s_1' \mid s_2'} \quad (tFlow)$$

$$\frac{\forall\, i \in I \quad m, t \gg g_i \xrightarrow{\delta} m, t+\delta \gg g_i' \quad timeOut(g_i, \delta, t) = \mathbf{false}}{m, t \gg \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t+\delta \gg \sum_{i \in I} g_i'; s_i} \quad (tPick1)$$

$$\frac{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta} m, t+\delta \gg \mathbf{0}}{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta); s + \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t+\delta \gg s} \quad (tPick2)$$

$$\frac{m, t \gg \mathbf{wait}_{\mathrm{u}}(t+\delta) \xrightarrow{\delta} m, t+\delta \gg \mathbf{0}}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t+\delta); s + \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t+\delta \gg s} \quad (tPick3)$$

execution, i.e. they do not consume time units. Consequently, rules for exit, assign, invoke, receive, if-then-else, sequence, flow and pick, are modified according to the new definition of configuration (i.e. adding the same clock before and after the transition). E.g., the rule for the sequence activity is modified as follows:

$$\frac{m, t \gg s_1 \xrightarrow{\alpha} m, t \gg s_1' \quad \alpha \notin \mathtt{Time}}{m, t \gg s_1; s_2 \xrightarrow{\alpha} m, t \gg s_1'; s_2} \quad (Sequence)$$

Let us briefly comment some rules of the operational semantics of timed WS-CALCULUS services. Rule (*tReceive*) permits time elapsing before the execution of the receive activity. Rules ($Wait_f\,1$) and ($Wait_f\,2$) permit to update the argument of $\mathbf{wait}_{\mathrm{f}}(\delta)$ until the timeout expires. Rules ($Wait_u\,1$) and ($Wait_u\,2$) permit time elapsing until the clock of the instance reaches the argument of $\mathbf{wait}_{\mathrm{u}}(t)$. Time elapsing in a sequence activity is governed by rule (*tSequence*). To synchronize time elapsing in a flow activity we use rule (*tFlow*). ($tPick_n$) rules determine that the time elapsing cannot make a choice within a pick activity, except when a timeout occurs.

The major modifications of the operational semantics for nets are represented by the following two rules:

$$\frac{\forall\, m, t \gg s \in C \quad m, t \gg s \xrightarrow{\delta} m, t+\delta \gg s'}{\{a ::_t C\} \rightarrowtail \{a ::_{t+\delta} C_\delta\}} \quad (tSync)$$

$$\frac{\emptyset, t \gg s \xrightarrow{?(r,o,\bar{w})} \emptyset, t \gg s' \qquad \emptyset \rhd (r, \bar{w} := \underline{a}', \bar{u}) = \langle m, \sigma \rangle \qquad \#R(*s \mid C, \_, \langle a', o, \bar{u} \rangle) = 1}{\{a ::_t *s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a ::_t *s \mid m, t \gg s' \cdot \sigma \mid C\}} \quad (Receive_S)$$

Rule (*tSync*) says that all the instances of a service are time-synchronized (i.e. time elapsing transitions

Table 10. WS-CALCULUS plus compensation

s	::=	$\ldots$	
	$\mid$	$[s : h]_\kappa$	(scope)
	$\mid$	$\mathbf{throw}(\phi)$	(fault generator)
	$\mid$	$\mathbf{undo}(\kappa) \mid \mathbf{undo}$	(compensate/compensate all)
	$\mid$	$\uparrow s \uparrow$	(protection)
h	::=		(handlers)
		$\mathbf{catch}(\phi)\{s\} : h$	(specific fault)
	$\mid$	$\mathbf{catchAll}\{s_{all}\} : \mathbf{comp}\{s_{cmp}\}$	(generic fault/compensation)
Δ	::=	$(\kappa_n \mapsto s) \mid (\kappa_n \mapsto nsc) \mid (\kappa_n \mapsto undone) \mid \Delta \circ \Delta$	(compensation function)

are synchronous). The notation C_δ indicates the set of components obtained from C by replacing each instance $m, t \gg s$ with $m, t + \delta \gg s'$. The last rule ($Receive_S$) permits to initiate the local clock of a new service instance.

5.3. Scopes and compensation handling

Similarly to WS-BPEL, WS-CALCULUS error handling is related to the notion of *compensation*, namely specific activities that attempt to reverse the effects of previously executed activities. We introduce here a variant of *Sagas* compensation protocol [GMS87]. In Table 10, we extend WS-CALCULUS syntax including, in particular, the scoping construct (or scope, for short) $[s : h]_\kappa$ used for grouping explicitly together activities identified by means of an unique scope identifier κ. A scope provides an optional list of *specific* fault handlers $\mathbf{catch}(\phi_1)\{s_1\} : \mathbf{catch}(\phi_2)\{s_2\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\}$, with $n \geq 0$, including possibly a *termination* handler $\mathbf{catch}(\phi_{exit})\{s_{exit}\}$, and followed by a *generic* fault handler $\mathbf{catchAll}\{s_{all}\}$ and by a *compensation* handler $\mathbf{comp}\{s_{cmp}\}$ (notably, $\mathbf{catchAll}\{s_{all}\}$ and $\mathbf{comp}\{s_{cmp}\}$ are, in order, at the end of the list h). Each $\mathbf{catch}(\phi)\{s\}$ is defined to intercept the specific kind of fault identified by ϕ. The *fault generator* $\mathbf{throw}(\phi)$ is used to signal an internal fault ϕ from within a scope. The fault handling behaviour is standard: in case of a fault signal ϕ, the corresponding fault handler $\mathbf{catch}(\phi)\{s\}$, if exists within the scope, is selected, otherwise the generic handler $\mathbf{catchAll}\{s_{all}\}$ is chosen. The *compensate* activity $\mathbf{undo}(\kappa)$ can be used to invoke a compensation associated with an inner scope identified by κ that has already completed successfully, i.e., without faulting. This activity can be invoked only from within a fault handler or another compensation handler. When the scope name is omitted, **undo** causes all inner scopes to execute their compensation handlers in reverse order. Such as in WS-BPEL, we consider illegal mixed usage of both activities **undo** and $\mathbf{undo}(\kappa)$ within a single scope (since the latter disables the availability of the former compensation activity). Moreover, we consider incorrect handlers containing scope constructs. We assume services observing the above conditions and having the following properties of *undo-correctness*: for each handler h_i of a scope containing $\mathbf{undo}(\kappa)$ (resp. **undo**) there exists at least an inner scope named κ.

To define the operational semantics of WS-CALCULUS compensations, we indicate the set of fault identifiers as $\mathbb{FID}$ and the set of scope identifiers as $\mathbb{SID}$. Moreover, we use $\mathbb{SID}_\mathbb{N}$ to denote the set of *indexed* scope

Table 11. Forced Termination $halt(\cdot)$ / Compensation stack push function $csp(\cdot,\cdot)$

$$halt(s_1; s_2) = halt(s_1) \quad halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2) \quad halt(\uparrow s \uparrow) = \uparrow s \uparrow$$

$$halt([s : h]_\kappa) = \uparrow halt(s); s' \uparrow \quad if \;\; \mathbf{catch}(\phi_{exit})\{s'\} \in h \;\; otherwise \;\; s' = s_{all}$$

$$csp(\mathbf{undo}, \kappa_n) = \uparrow \mathbf{undo}; \mathbf{undo}(\kappa_n) \uparrow \qquad csp(\mathbf{undo}(\kappa), \kappa_n) = \uparrow \mathbf{undo}(\kappa); \mathbf{undo}(\kappa_n) \uparrow$$

$$csp(\mathbf{undo}(\kappa'), \kappa_n) = \mathbf{undo}(\kappa') \quad \forall \kappa' \in \mathbb{SID} : \kappa' \neq \kappa \qquad csp(\mathbf{undo}(\kappa'_m), \kappa_n) = \mathbf{undo}(\kappa'_m) \quad \forall \kappa'_m \in \mathbb{SID}_{\mathbb{N}}$$

$$csp(\mathbf{catch}(\phi)\{s\} : h, \kappa_n) = \mathbf{catch}(\phi)\{csp(s, \kappa_n)\} : csp(h, \kappa_n)$$

$$csp(\mathbf{catchAll}\{s\} : \mathbf{comp}\{s'\}, \kappa_n) = \mathbf{catchAll}\{csp(s, \kappa_n)\} : \mathbf{comp}\{csp(s', \kappa_n)\}$$

identifiers. Indexed identifiers are introduced because a scope may occur within a recursive definition, and the invocation of the installed compensation handlers in the successive iterations must be performed in reverse order. The extended semantics of WS-CALCULUS makes use of *compensation functions*, ranged over by Δ, Δ', to map each (indexed) identifier κ_n to the associated compensation handlers. Specific symbols *nsc* and *undone* represent, respectively, scopes *not successfully completed* and scopes in which the compensation handler has been already invoked. We assume that each compensation function Δ is initially defined so that we have $\Delta(\kappa_n) = nsc$, for all $\kappa_n \in \mathbb{SID}_{\mathbb{N}}$. Moreover, we indicate with $\Delta \circ \Delta'$ the compensation function obtained by updating Δ with Δ'.

As prescribed by WS-BPEL, fault handling or compensation activities must be treated as protected behaviours by the semantics of the termination activity **exit**. For this reason, we further extend the syntax by introducing a *protection* construct $\uparrow s \uparrow$ inspired by StAC$_i$ (a language for modelling long-running business transactions, [BF04]). The extended semantics of the activity **exit** exploits an auxiliary function $halt(s)$ that removes all the activities from the given service s, with the exception of (recursively nested) protected activities. In upper part of Table 11, we show the only cases in which $halt(s)$ is different from $\mathbf{0}$. In particular, rule $halt(\uparrow s \uparrow) = \uparrow s \uparrow$ states that protected activities are unaffected by the function $halt(\_)$; and rule $halt([s : h]_\kappa) = \uparrow halt(s); s' \uparrow$ states that force termination signals can be intercepted and be handled as being fault signals.

Extended Semantics. The extended semantics for compensations is given in terms of transitions between configurations, written $m \vdash (s, \Delta) \xrightarrow{\alpha} (s', \Delta')$, where Δ and Δ' stand for the above-mentioned compensation functions from scope identifiers to compensation handlers. The reduction relation is presented in Table 12, where the label α is now generated by the following grammar:

$$\alpha \;::=\; \ldots \;\mid\; \tau \;\mid\; \phi \;\mid\; \kappa_n$$

To comment on rules in Table 12, let h be the following list of handlers:

$$h = \mathbf{catch}(\phi_1)\{s_1\} : \mathbf{catch}(\phi_2)\{s_2\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : \mathbf{catchAll}\{s_{all}\} : \mathbf{comp}\{s_{cmp}\}$$

Fault generation and fault handling are described by rules (*Throw*), (*Fault*$_1$) and (*Fault*$_2$): when a fault signal occurs within a scope, we force the termination of the inner service by protecting the execution of the resulting service composed with the fault handling. A protection activity performs the same action performed by the

Table 12. Operational semantics: compensation

$$\frac{\phi \in \mathbb{FID}}{m \vdash (\mathbf{throw}(\phi), \Delta) \xrightarrow{\phi} (\mathbf{0}, \Delta)}\ (\mathit{Throw}) \qquad \frac{m \vdash (s, \Delta) \xrightarrow{\alpha} (s', \Delta')}{m \vdash (\uparrow s \uparrow, \Delta) \xrightarrow{\alpha} (\uparrow s' \uparrow, \Delta')}\ (\mathit{Protect})$$

$$\frac{m \vdash (s, \Delta) \xrightarrow{\phi_i} (s', \Delta) \quad \phi_i \in h}{m \vdash ([s : h]_\kappa, \Delta) \xrightarrow{\tau} (\uparrow halt(s'); s_i \uparrow, \Delta)}\ (\mathit{Fault}_1) \qquad \frac{m \vdash (s, \Delta) \xrightarrow{\phi} (s', \Delta) \quad \phi \notin h}{m \vdash ([s : h]_\kappa, \Delta) \xrightarrow{\tau} (\uparrow halt(s'); s_{all} \uparrow, \Delta)}\ (\mathit{Fault}_2)$$

$$\frac{\Delta(\kappa_n) = nsc \quad \nexists m \in \mathbb{NAT} : m < n \wedge \Delta(\kappa_m) = nsc}{m \vdash ([\mathbf{0} : h]_\kappa, \Delta) \xrightarrow{\kappa_n} (\mathbf{0}, \Delta \circ (\kappa_n \mapsto s_{cmp}))}\ (\mathit{ScopeCmp})$$

$$\frac{m \vdash (s, \Delta) \xrightarrow{\alpha} (s', \Delta') \quad \alpha \notin (\mathbb{FID} \cup \mathbb{SID}_\mathbb{N})}{m \vdash ([s : h]_\kappa, \Delta) \xrightarrow{\alpha} ([s' : h]_\kappa, \Delta')}\ (\mathit{ScopeExc}_1) \qquad \frac{m \vdash (s, \Delta) \xrightarrow{\kappa'_n} (s', \Delta') \quad \kappa'_n \in \mathbb{SID}_\mathbb{N}}{m \vdash ([s : h]_\kappa, \Delta) \xrightarrow{\tau} ([s' : csp(h, \kappa'_n)]_\kappa, \Delta')}\ (\mathit{ScopeExc}_2)$$

$$\frac{s \in \{\mathbf{undo}, \mathbf{undo}(\kappa)\} \quad \mathit{with}\ \kappa \in \mathbb{SID}}{m \vdash (s, \Delta) \xrightarrow{\tau} (\mathbf{0}, \Delta)}\ (\mathit{Undo}) \qquad \frac{\kappa_n \in \mathbb{SID}_\mathbb{N} \quad \Delta(\kappa_n) = s_{cmp}}{m \vdash (\mathbf{undo}(\kappa_n), \Delta) \xrightarrow{\tau} (s_{cmp}, \Delta \circ (\kappa_n \mapsto \mathit{undone}))}\ (\mathit{Cmp})$$

$$\frac{\kappa_n \in \mathbb{SID}_\mathbb{N} \quad \Delta(\kappa_n) = \mathit{undone}}{m \vdash (\mathbf{undo}(\kappa_n), \Delta) \xrightarrow{\phi_{repeatedCmp}} (\mathbf{0}, \Delta)}\ (\mathit{RptCmp})$$

protected service such as in (*Protect*). Rule (*ScopeCmp*) says that a successfully completed scope signals its completion and installs the compensation handler into the compensation function. By rule ($\mathit{ScopeExc}_1$) permits to perform any action within the scope except for fault signals and scope completion signals. Scope completion signals are consumed and handled by rule ($\mathit{ScopeExc}_2$) by exploiting a *compensation stack push* function, denoted by $csp(h, \kappa_n)$ with $\kappa_n \in \mathbb{SID}_\mathbb{N}$. This function replaces each **undo** construct (resp. $\mathbf{undo}(\kappa)$) occurring in h with $\uparrow \mathbf{undo}; \mathbf{undo}(\kappa_n) \uparrow$ (resp. $\uparrow \mathbf{undo}(\kappa); \mathbf{undo}(\kappa_n) \uparrow$). It is defined inductively on the syntax of h and the most relevant cases are shown in the lower part of Table 11. Rule (*Undo*) says that **undo** and $\mathbf{undo}(\kappa)$ produce a silent action since their role is only to be the prefix of sequence of $\mathbf{undo}(\kappa_n)$ activities. Moreover, this rule says that invoking a compensation handler that has not been installed (the scope is not successfully completed yet) is equivalent to the empty activity. Finally, rule (*Cmp*) permits to perform a specified compensation handler, while rule (*RptCmp*) signals the fault $\phi_{repeatedCmp}$ when a compensation handler is invoked more than once.

6. Concluding remarks

We have set a formal semantics framework for web services orchestration languages, and particularly for WS-BPEL. We have introduced WS-CALCULUS, a foundational language specifically designed for modelling interactions among web services. We also have presented an illustrative example borrowed from the official specification of WS-BPEL. Finally, we have extended WS-CALCULUS to also deal with such constructs as flow graphs, timed activities, scopes and compensation handling, thus giving a more complete semantic account of WS-BPEL executable processes.

Related work. The major contribution of our work is the formal modelling of different aspects of WS-BPEL, such as multiple start activities, receive conflicts, routing of correlated messages, interactions among different web services. Since we wanted to study those problems arising when executing WS-BPEL processes, then we have focused on service orchestration rather than on service choreography, that instead provides a means to describe service interactions in a top-view way (these aspects have been considered in e.g. [BGG+05, CHY07]). The mechanism of correlation sets was first investigated in [Vir04], that however only consider interaction of different instances of a single business process.

Several formal semantics of WS-BPEL were proposed in the literature. Many of these efforts aim at formalizing a *complete* semantics for WS-BPEL using Petri nets [OvB+05, Loh08] or workflow [CPM06], but do not cover such dynamical aspects as service instantiation and message correlation. Other works [GXSZ05, GLG+06] using process calculi focus instead on small and relatively simple subsets of WS-BPEL. Another bunch of related works (e.g. [LZ05a, ML06]) are targeted to formalize the semantics of WS-BPEL by encoding parts of the language into more foundational orchestration languages. Our work differs for the fact that, by electing WS-BPEL as starting point, we distill a core model of interaction whose dynamical aspects are fully taken into account. As a second contribution, we have extended the WS-CALCULUS to incorporate such other relevant constructs of WS-BPEL as synchronous interactions, flow graphs, timed activities, scopes and compensation handling, that should be considered for modelling the semantics of full-blown orchestration languages. We have also shown that synchronous interactions and flow graphs can be easily encoded in WS-CALCULUS, while the semantics of the remaining constructs is given by extending the operational semantics of WS-CALCULUS. This way, we have defined an operational semantics for WS-BPEL *executable* processes (without any syntactical restriction).

As we have already said, many research efforts are addressed to define clean semantic models and rigorous methodological foundations for WS-BPEL programs. Recently, a very general and flexible framework for error recovery has been introduced in [GLMZ08]; this framework extends [GLG+06], the language for service composition closest to WS-CALCULUS, with dynamic compensation and modelling in particular the dependency between fault handling and the request-response communication pattern.

Some other relevant related works are [BMM05, BBF+05, BLZ03]. In the first two, the authors propose a formal approach to model compensation in transactional calculi and present a detailed comparison with [BF04]. The third is an extension of the asynchronous π-calculus with long-running (scoped) transactions. The language has a scope construct which plays a role similar to the scope activity presented in our extension of WS-CALCULUS, but it is not aimed at capturing the order in which compensations should be activated. On the contrary, the semantics of compensations we propose in this work faithfully captures the intended semantics of WS-BPEL, thus for example compensations are activated in the reverse order with respect to the order of completion of the original scopes.

Most of these formalisms, however, do not model the different aspects of currently available WS-BPEL features in their completeness. One such aspect is represented by *timed activities* that are frequently exploited in service orchestration and are typically used for handling timeouts. For example, in WS-BPEL, timeouts

turn out to be essential for dealing with service transactions or with message losses. Thus, a service process could wait a callback message for a certain amount of time after which, if no callback has been received, it invokes another operation or throws a fault. However, only a few process calculi for modelling WS-BPEL programs deal with timed activities. In particular, [LZ05a, LZ05b] introduce $\texttt{web}\pi$, a timed extension of the π-calculus tailored to study 'web transactions'. In $\texttt{web}\pi$ the notion corresponding to a *service engine* is that of 'machines'. Single machines possess their own clock that is not synchronised with the clock of other machines. However, one of the main and characteristic concerns of $\texttt{web}\pi$ is the *transaction construct* $\langle\!|P\,;\,Q|\!\rangle_x^n$ where P and Q are the body and the compensation, respectively, n indicates the deadline, and x is the name of the transaction. This very expressive construct (putting together time aspects with transactional mechanisms) facilitates the modelling of long running transactions and the encoding of transactional constructs such as the *scope activity* of WS-BPEL. Related work like [GHZ+06, GXSZ06] present a timed calculus based on a more general notion of time, and an approach to verify WS-BPEL specifications with compensation/fault constructs. [KCM06] proposes a general purpose task orchestration language that manages timeouts as signals returned by dedicated services after some specified time intervals. Differently from our timed-extension of WS-CALCULUS presented in this work, all these formalisms, do not take into account such fundamental aspects of web services as service instantiation and correlation.

References

[BBF+05] R. Bruni, M. Butler, C. Ferreira, C.A.R. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proceeding of CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.

[BF04] M.J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proceeding of COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.

[BGG+05] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *Proceeding of ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2005.

[BJK02] A. Brown, S. Johnston, and K. Kelly. Using service-oriented architecture and component-based development to build web service applications. Technical report, Rational Software Corp., 2002.

[BLZ03] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proceeding of FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[BMM05] R. Bruni, H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proceeding of POPL*, pages 209–220. ACM, 2005.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001. Available at `http://www.w3.org/TR/wsdl/`.

[CHY07] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proceeding of ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[CPM06] W.R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in orc. In *Proceeding of COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

[GHZ+06] P. Geguang, Z. Huibiao, Q. Zongyan, W. Shuling, Z. Xiangpeng, and H. Jifeng. Theoretical foundations of scope-based compensable flow language for web service. In *Proceeding of FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2006.

[GLG+06] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *Proceeding of ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[GLMZ08] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations. In *Proceedings of ACSD*. IEEE, 2008. To appear.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, pages 249–259. ACM Press, 1987.

[GXSZ05] P. Geguang, Z. Xiangpeng, W. Shuling, and Q. Zongyan. Semantics of BPEL4WS-like fault and compensation handling. In *Proceeding of FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2005.

[GXSZ06] P. Geguang, Z. Xiangpeng, W. Shuling, and Q. Zongyan. Towards the semantics and verification of BPEL4WS. *Electronic Notes in Theoretical Computer Science*, 151(2):33–52, 2006.

[KCM06] D. Kitchin, W.R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *Proceeding of CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[Loh08] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Proceeding of WSFM*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2008.

[LZ05a] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proceeding of FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[LZ05b] C. Laneve and G. Zavattaro. web-pi at work. In *Proceeding of TGC*, volume 3705 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2005.

[ML06] M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.

[OAS07] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/`.

[OvB+05] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in WS-BPEL (revised version). Technical report, BPM Center Report, 2005. Available at `http://www.BPMcenter.org`.

[Vir04] M. Viroli. Towards a formal foundational to orchestration languages. In *Proceeding of WSFM*, volume 105, pages 51–71, 2004.